The Journal of Systems and Software 116 (2016) 162–176

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Systematic scalability assessment for feature oriented multi-tenant services

Davy Preuveneers*, Thomas Heyman, Yolande Berbers, Wouter Joosen

iMinds-DistriNet, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

ARTICLE INFO

Article history:
Received 9 October 2014
Revised 11 December 2015
Accepted 14 December 2015
Available online 22 December 2015

Keywords:
Distributed systems
Scalability
Tool support

ABSTRACT

Recent software engineering paradigms such as software product lines, supporting development techniques like feature modeling, and cloud provisioning models such as platform and infrastructure as a service, allow for great flexibility during both software design and deployment, resulting in potentially large cost savings. However, all this flexibility comes with a catch: as the combinatorial complexity of optional design features and deployment variability increases, the difficulty of assessing system qualities such as scalability and quality of service increases too. And if the software itself is not scalable (for instance, because of a specific set of selected features), deploying additional service instances is a futile endeavor. Clearly there is a need to systematically measure the impact of feature selection on scalability, as the potential cost savings can be completely mitigated by the risk of having a system that is unable to meet service demand.

In this work, we document our results on systematic load testing for automated quality of service and scalability analysis. The major contribution of our work is tool support and a methodology to analyze the scalability of these distributed, feature oriented multi-tenant software systems in a continuous integration process. We discuss our approach to select features for load testing such that a representative set of feature combinations is used to elicit valuable information on the performance impact and feature interactions. Additionally, we highlight how our methodology and framework for performance and scalability prediction differs from state-of-practice solutions. We take the viewpoint of both the tenant of the service and the service provider, and report on our experiences applying the approach to an industrial use case in the domain of electronic payments. We conclude that the integration of systematic scalability tests in a continuous integration process offers strong advantages to software developers and service providers, such as the ability to quantify the impact of new features in existing service compositions, and the early detection of hidden feature interactions that may negatively affect the overall performance of multi-tenant services.

1. Introduction

Feature oriented software development, cloud computing and multi-tenancy are triggering a tremendous shift in the software systems landscape. Software product line (SPL) oriented development methods and feature modeling allow customers to pay only for the features they need, resulting in a potentially large reduction of software costs. Cloud based deployment environments allow customers to pay only for the computational resources they need, resulting in a potentially large reduction of operational costs. This operational cost reduction is pushed even further in multi-tenant software as a service (SaaS) applications, where all tenants share resources by using the same instance of the SaaS application. In order to maintain flexibility and be able to cater to varying tenant requirements, dynamic software product lines are often used, where tenant specific customization of the SaaS application is enforced at runtime. However, all this flexibility comes with a catch: As the combinatorial complexity of feature and deployment variability increases, the difficulty of assessing system qualities such as scalability and quality of service increases too. And if the software itself is not scalable (for instance, because of a specific set of selected features), deploying additional service instances is a futile endeavor.

There are two stakeholders in particular to whom (unanticipated) feature interactions and their impact on scalability and quality of service are important: the service customer (i.e. the entity that acquires a product from a software product line, or the tenant in a multi-tenant system) and the service provider (i.e. the

* Corresponding author. Tel.: +3216327853.
E-mail address: davy.preuveneers@cs.kuleuven.be (D. Preuveneers).

http://dx.doi.org/10.1016/j.jss.2015.12.024
0164-1212/

owner of a software product line, or the manager of a multi-tenant system). Service customers need to know what the cost of selecting a new feature is on overall service scalability, as they want to avoid rendering the system unable to scale up, resulting in an unresponsive system and frustrated end users. Service providers need to know what tenants can be hosted on the same machines without unanticipated interactions in virtualized environments. Deploying a new customer's service (and associated features) in a virtualized environment could interact with, and impair, the scalability of another customer's service, which would equally frustrate that customer.

Clearly there is a need to systematically measure the impact of feature selection on scalability, as the potential cost savings can be completely mitigated by the risk of having a system that is unable to meet service demand. While a large body of work exists that focuses on specific parts of this scalability problem, not much work has been done to study whether it is feasible to quantify the scalability and performance of such a complex distributed system in a practical way. In this work, we document a holistic approach on systematic load testing for automated quality of service and scalability analysis. The major contribution of our work is tool support and a methodology for scalability analysis of these distributed, feature oriented software systems throughout a continuous integration process. We take the viewpoint of both the service customer (i.e. the tenant) and the service provider, and report on our experiences applying the approach to an industrial use case in the domain of electronic payments. We conclude that it is possible to integrate systematic scalability tests in a continuous integration process, which allows early detection of feature interactions that negatively impact performance.

This paper is structured as follows. In Section 2, we make explicit what we mean by scalability and quality of service, and we summarize the current state of the art of scalability and quality of service assessment. In Section 3, we introduce a case study in the domain of e-payment, as well as an implementation, to illustrate the problem of scalability assessment in this context. Additionally, we introduce our system for systematic scalability assessment. In Section 4, we introduce the framework and methodology we use to perform systematic scalability assessment on multi-tenant feature oriented systems. In Section 5, we document our results of applying the framework to the case study. The work is discussed in Section 6. We conclude in Section 7.

2. Background and related work

We give a brief overview of scalability in Section 2.1, and overview dynamic software product lines and feature modeling in Section 2.2. We summarize related work on performance modeling and prediction in Section 2.3.

2.1. Scalability

Assessing system performance and scalability is a practice that cross cuts many levels of abstraction, ranging from low-level benchmarks of execution environments and embedded software, to high-level distributed systems and business process benchmarks. For instance, Guthaus et al. (2001) perform benchmarking on embedded programs and provide a comparison with the industry standard benchmark suite SPEC2000. Ghosh et al. (2005) analyze the performance of WiMax networks. Uskov (2012) provides a comprehensive study of the performance of authentication and encryption algorithms for virtual private networking. Rashwan et al. (2012) study the performance of message authentication codes for mobile networks, for both residence time and power consumption. Dayarathna and Suzumura (2013) document their results of comparing the performance of three complex event processing engines via benchmarking. Carvalho and Pereira (2010) document a method to analyze scalability of running systems from the data center viewpoint, by only measuring CPU utilization.

This work considers the scalability of large, multi-tenant distributed software systems. The scalability of a system is often defined as its ability to handle increasing user load. When faced with a user load of p concurrent users that issue a certain volume of requests per second, a service will be able to successfully handle a specific fraction of p, called its throughput, and denoted by $X(p)$. In the ideal case, a service can handle the requests generated by all users concurrently, or $X(p) = p$. That situation, also referred to as linear scalability, only tends to hold in real systems for low values of p, i.e. low load situations. For increasing loads, however, $X(p) < p$, as a system will not be able to successfully handle the requests generated by all users concurrently, and some requests will have to wait or, in extreme cases, be dropped. To find $X(p)$, we can simulate p concurrent users that issue requests at a fixed rate, and measure $X(p)$. Based on this data, we can calculate the capacity ratio $C(p) = X(p)/X(1)$, which is the ratio of the throughput of the system for a load of p, compared to the baseline of its throughput for a load of 1. The relative capacity curve $C(p)$ is a good indicator for how much the observed behavior diverges from the ideal linear scalability $C_L(p) = p$. The closer $C(p)$ is to $C_L(p)$, the better the scalability of that system configuration.

There are a number of models that attempt to quantify the capacity C of a system. One of these models, the universal scalability law (USL) (Gunther, 1993; Gross et al., 2013), takes into account both the serial nature of the workload of that system (i.e. how much of the workload can be parallelized in theory) and coherency costs (i.e. the costs incurred when waiting for data to become consistent between different collaborating processes). The universal scalability law quantifies capacity in function of user load as:

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)}$$

Here, κ denotes the impact of coherency on the system performance, and σ denotes the serial fraction, which is the fraction of the workload that cannot be parallelized. When the coherency factor κ is negligible, the maximum performance of the system is bounded only by the serial fraction, and the model reduces to Amdahl's Law (Amdahl, 1967). When κ is non zero, the performance model of a system will have a specific maximum, achieved for a load $p^* = \lfloor\sqrt{(1+\sigma)/\kappa}\rfloor$. Beyond p^*, the throughput of a system will decrease. An illustration of a scalability curve according to the Universal Scalability Law, and a comparison to Amdahl's Law, is given in Fig. 1. We can find values for σ and κ for a specific service deployment by performing linear regression on measured values for $C(p)$ (Gunther, 2007).

So what happens to a request when the load is sufficiently high that $C(p) \ll p$? When the system remains stable and does not drop requests, the residence time of those requests (i.e. the time between issuing a request and receiving an answer) will start to grow exponentially. While the service capacity considers the business view (i.e. *How many servers do I need to handle this many concurrent users?*), residence time considers the end user's perspective (i.e. *How long do I have to wait before my request is handled?*), and is therefore an equally important aspect of scalability to consider. However, only knowing the average residence time does not suffice. In the case of quality of service policies, which are usually expressed in function of X% of requests that need to be handled within Y (milli-) seconds, we need to know the overall statistical distribution of the residence times.

To measure values for $C(p)$ and obtain residence time distributions, load testing frameworks simulate users by generating actual requests. There are a number of load testing frameworks in